46장. 제너레이터와 async/await

46.1 제너레이터란?

- **제너레이터(generator)**

  : 코드 블록 실행 일시 중지했다가 필요한 시점에 재개할 수 있는 특수한 함수

제너레이터와 일반 함수의 차이

1. **제너레이터 함수는 함수 호출자에게 함수 실행의 제어권을 양도할 수 있다.**

   일반 함수 - 함수를 호출한 이후 함수 실행을 제어할 수 없음

2. **제너레이터 함수는 함수 호출자와 함수의 상태를 주고받을 수 있다.**

   일반 함수 - 함수가 실행되는 동안 외부에서 내부로 값을 전달하여 상태 변경 불가능

3. **제너레이터 함수를 호출하면 제너레이터 객체를 반환한다.**

   일반 함수 - 함수 코드를 실행하고 값을 반환

46.2 제너레이터 함수의 정의

1. 제너레이터 함수는 `function*` 키워드로 선언.

   그리고 하나 이상의 yield 표현식을 포함

2. `*(애스터리스크)`의 위치는 키워드, 함수 이름 사이라면 어디든지 상관 X

```
function* genFunc() { yield 1; }
function * genFunc() { yield 1; }
function *genFunc() { yield 1; }
function*genFunc() { yield 1; }
```

3. 제너레이터 함수는 화살표 함수로 정의 ❌

4. 제너레이터 함수는 new 연산자와 함께 생성자 함수로 호출 ❌

46.3 제너레이터 객체

- **제너레이터 함수를 호출하면 일반 함수처럼 함수 코드 블록을 실행하는 것이 아니라, 제너레이터 객체를 생성해 반환.**

  제너레이터 함수가 반환한 제너레이터 객체는 이터러블(iterable)이면서 동시에 이터레이터(iterator)

  ➡️ 제너레이터 객체는 Symbol.iterator 메서드를 상속받는 이터러블이면서 value, done 프로퍼티 를 갖는 이터레이터 리절트 객체를 반환하는 next 메서드를 소유

1. next 메서드 호출

- yield 표현식까지 코드 블록 실행하고, 값을 value 프로퍼티 값으로 false를 done 프로퍼티 값으로 갖는 이터레이터 리절트 객체 반환

2. return 메서드 호출

- 인수로 전달받은 값을 value로, true를 done 프로퍼티 값으로 갖는 이터레이터 리절트 객체 반환

```
function* genFunc(){
    try {
        yield 1;
        yield 2;
        yield 3;
    } catch (e) {
      console.error(e);
    }
}

const generator = genFunc();

console.log(generator.next());  // { value: 1, done: false }
console.log(generator.return('End!'));  // { value: "End!", done: true }
```

3. throw 메서드 호출

- 인수로 전달받은 에러 발생시키고 undefined를 value 프로퍼티 값으로, done은 true 갖는 이터레이터 리절트 객체 반환

```
function* genFunc(){
    try {
        yield 1;
        yield 2;
        yield 3;
    } catch (e) {
       console.error(e);
    }
}

const generator = genFunc();
```

```
console.log(generator.next());    // { value: 1, done: false }
console.log(generator.throw('Error!'));   // { value: undefined, done: true }
```

46.4 제너레이터의 일시 중지와 재개

1. **yield 키워드는 제너레이터 함수의 실행을 일시 중지시키거나 yield 키워드 뒤에 오는 표현식의 평가 결과를 제너레이터 함수 호출자에게 반환한다.**

```
function* genFunc(){
    yield 1;
    yield 2;
    yield 3;
}

const generator = genFunc();

console.log(generator.next());    // { value: 1, done: false }
console.log(generator.next());    // { value: 2, done: false }
console.log(generator.next());    // { value: 3, done: false }
console.log(generator.next());    // { value: undefined, done: true }
// 제너레이터 함수가 끝까지 실행되었기 때문에 done: true
```

2. **next 메서드 호출하면 yield 표현식까지 실행되고 일시중지(suspend)된다.**

   이때 함수의 제어권이 호출자로 양도(yield)된다.

3. **제너레이터 객체의 next 메서드에 전달한 인수는,**

   제너레이터 함수의 yield 표현식을 할당받는 변수에 할당된다.

```
function* genFunc() {
    // x 변수의 값은 next 메서드가 두 번째 호출될 때 결정됨.
    const x = yield 1;

    const y = yield(x + 10);

    return x + y;
}

const generator = genFunc(0);

// 처음 호출하는 next 메서드에는 인수 전달 X
// 전달했더라도 무시됨.
let res = generator.next();
```

```
console.log(res);    // { value: 1, done: false }

res = generator.next(10);
console.log(res);    // { value: 20, done: false }

res = generator.next(20);
console.log(res);    // { value: 30, done: true }
```

📝결론

제너레이터 함수는 next 메서드 + yield 표현식을 통해 함수 호출자, 함수의 상태를 주고 받을 수 있다.

46.5 제너레이터의 활용

46.5.1 이터러블의 구현

1. 제너레이터 함수를 사용

```
// 무한 이터러블을 생성하는 제너레이터 함수
const infiniteFibonacci = (function* () {
    let [pre, cur] = [0, 1];

    while(true){
            [pre, cur] = [cur, pre + cur];
            yield cur;
    }
}());

// infiniteFibonacci는 무한 이터러블이다.
for(const num of infiniteFibonacci){
    if ( num > 10000 ) break;
    console.log(num);  // 1 2 3 5 8 13 21 34 ....
}
```

46.5.2 비동기 처리

1. 제너레이터 함수 : next 메서드 + yield 표현식으로 함수 호출자, 함수의 상태를 주고받을 수 있다.

   → 프로미스를 사용한 비동기 처리를 동기 처리처럼 구현 가능

```
// node-fetch는 Node.js 환경에서 window.fetch 함수를 사용하기 위한 패키지
// 브라우저 환경에서 이 예제를 실행한다면 아래 코드 필요 X
// https://github.com/node-fetch/node-fetch
const fetch = require('node-fetch');
```

```
// 제너레이터 실행기
const async = generatorFunc => {
    const generator = generatorFunc(); // ②

    const onResolved = arg => {
        const result = generator.next(arg); // ⑤

        return result.done
            ? result.value // ⑨
            : result.value.then(res => onResolved(res)); // ⑦
    };
    return onResolved;  // ③
};

(async(function* fetchTodo() {  // ①
    const url = 'https://jsonplaceholder.typicode.com/todos/1';

    const response = yield fetch(url); // ⑥
    const todo = yield response.json(); // ⑧
    console.log(todo);
    // { userId: 1, id: 1, title: 'delectus aut autem', completed: false}
})()); // ④
```

2. co 라이브러리 사용

```
const fetch = require('node-fetch');
const co = require('co');

(async(function* fetchTodo() {
    const url = 'https://jsonplaceholder.typicode.com/todos/1';

    const response = yield fetch(url);
    const todo = yield response.json();
    console.log(todo);
    // { userId: 1, id: 1, title: 'delectus aut autem', completed: false}
})());
```

46.6 async/await

1. 제너레이터 사용해서 비동기 처리를 동기 처리처럼 동작하도록 구현

   → 코드가 엄청 장황해지고 가독성도 나빠짐

   ➡️ ES8에서는 더 간단하고 가독성 좋은 **async/await** 도입

2. 위 예제를 async/await로 다시 구현

```
const fetch = require('node-fetch');
const co = require('co');

async function fetchTodo() {
    const url = 'https://jsonplaceholder.typicode.com/todos/1';

    const response = await fetch(url);
    const todo = await response.json();
    console.log(todo);
    // { userId: 1, id: 1, title: 'delectus aut autem', completed: false}
}

fetchTodo();
```

46.6.1 async 함수

1. await 키워드는 반드시 async 함수 내부에서 사용(언제나 프로미스를 반환)

```
// async 함수 선언문
async function foo(n) { return n }
foo(1).then(v => console.log(v)); // 1

// async 함수 표현식
const bar = async function(n) { return n };
bar(2).then(v => console.log(v)); // 2

// async 화살표 함수
const baz = async n => n;
baz(n).then(v => console.log(v)); // 3

// async 메서드
const obj = {
  async foo(n) { return n; }
};
obj.foo(4).then(v => console.log(v)); // 4

// async 클래스 메서드
class MyClass {
  async bar(n) { return n; }
}
const myClass = new MyClass();
myClass.bar(5).then(v => console.log(v)); // 5
```

46.6.2 await 키워드

1. await 키워드는 프로미스가 settled 상태(비동기 처리가 수행된 상태)가 될 때까지 대기하다가 settled 상태가 되면 프로미스가 resolve한 처리 결과를 반환

```
const getGithubUserName = async id => {
  const res = await fetch(`https://api.github.com/users/${id}`);
  const { name } = await res.json();
  console.log(name); // seunghyune
};

getGithubUserName('SeungHyune')
```

2. `await` 키워드는 프로미스가 `settled` 상태가 될 때까지 대기하다

   프로미스가 `settled` 상태가 되면 프로미스가 resolve한 결과를 res 변수에 할당

   → await 키워드는 다음 실행을 일시 중지시켰다가 프로미스가 settled 상태가 되면 다시 재개

46.6.3 에러 처리

- 비동기 처리를 위한 콜백 패턴의 단점 중 가장 심각한 것은 **에러처리가 곤란하다는 것**

1. 에러는 호출자 방향으로 전파됨.

   하지만 콜백 함수를 호출한 것은 비동기 함수가 아니기 때문에 `try...catch` 문을 사용해 에러를 캐치할 수 없음.

2. async/await 에러처리 ⇒ `try…catch 문`

```
const fetch = require('node-fetch');

const foo = async () => {
  try {
    const url = 'https://wrong.url';

    const response = await fetch(url);
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error); // TypeError : Failed to fetch
  }
}
foo();
```

- 예제의 foo 함수의 catch 문은 HTTP 통신에서 발생한 네트워크 에러뿐 아니라,

  try 코드 블록 내의 모든문에서 발생한 일반적인 에러까지 모두 캐치할 수 있다.

async 함수 내에서 catch 문을 사용해서 에러 처리를 하지 않으면 async 함수는 발생한 에러를 reject하는 프로미스를 반환합니다.